John von Neumann Institute for Computing

A Run-time Reconfigurable Cache Architecture

Fabian Nowak, Rainer Buchty, Wolfgang Karl

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr,
F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 757-766, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

© 2007 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume38

A Run-time Reconfigurable Cache Architecture

Fabian Nowak, Rainer Buchty, and Wolfgang Karl

Institut für Technische Informatik (ITEC)
Universität Karlsruhe (TH)
76128 Karlsruhe, Germany
E-mail: {nowak, buchty, karl}@ira.uka.de

Cache parameters for CPU architectures are typically defined for a best overall match regarding the targeted field of applications. However, such may hinder high-performance execution of single applications and also does not account for cache access phases as occurring in long-running applications, e.g. from field of high-performance computing. It has been shown that matching the cache parameters to a running application results in both application speed-up and increased energy efficiency. The aim of the presented work is to create a versatile, reconfigurable cache hardware infrastructure for cache performance analysis. Such an infrastructure enables real-time monitoring of running applications and therefore is able to better trace a running application's behaviour compared to off-line analysis of a more or less reduced trace. In this paper, we will address the problems and side-effects of a run-time reconfigurable cache architecture, to which we present appropriate solutions. We will also give an outline of the upcoming hardware prototype.

A Introduction and Motivation

In traditional hardware architectures, hardware parameters are defined with respect to given application scenarios. For general purpose architectures, this results in a compromise-based architecture serving best exactly this application set, but possibly unsuitable for any other application set.

This is especially true for the cache subsystem and may hinder the execution of high-performance computing applications which reportedly show distinct phases of certain cache access behaviour[1]. Such applications would greatly benefit from a reconfigurable cache infrastructure which can be reconfigured on-the-fly in order to match the current requirements. Using dedicated cache architectures can also improve energy efficiency[2–4].

The presented work aims at providing a new means for on-line cache performance analysis by creating an adjustable, versatile hardware infrastructure. In contrast to off-line analysis, such an infrastructure will enable real-time monitoring of running applications instead of off-line analysis of a more or less reduced trace.

A substantial technology for exploiting on-line cache analysis is reconfiguration of the cache infrastructure, which introduces certain problems that we will address in this paper. After identifying possible cache reconfiguration points, we will prove the feasibility of such a system and explain how to overcome the individual problems. Finally, we will outline the upcoming hardware prototype implementation of our approach providing insight into our ongoing research.

Such an architecture is well suited for cluster architectures with heterogeneous, FPGA-based nodes.

B Related Work

Much effort has already been put into the analysis of applications' cache behaviour[5,6] and the development of a configurable cache infrastructure[2,3,7–9]. In their approach, applications are measured with respect to their individual behaviour and the cache infrastructure is configured with respect to a single running application. During run-time of an application, the cache configuration remains unchanged. As a result of that, no problematic data loss (cache flush) resulting from cache reconfigurations occurs because reconfiguration never takes place while an application is under execution.

The introduction of a switched L1-/L2-/L3-hierarchy that distributes on-chip memory to the different cache levels based on the CPI value of a program phase[4] suffers from the big drawback that measuring the CPI value requires changes in the internal architecture of the executing hardware, and it is even shown[10] that more sophisticated metrics are needed for phase detection.

On a further side note, benchmarking of applications in such environments is typically done off-line; with changed or new applications the benchmarking has to be re-done offline in order to determine the corresponding suitable cache configuration, which can then be used to initially configure the cache for the following application.

In contrast, our work focuses on run-time reconfiguration possibility without inducting the need for changes in the processing unit. Hence, much effort was put into the development of an efficient reconfigurable cache architecture, which allows on-the-fly reconfiguration of typical cache parameters while avoiding unnecessary cache flushes.

C Cache Parameters and Reconfiguration

Caches[11] are defined by their overall size, associativity, line size, replacement, write-back and allocation strategy.

It can be observed that cache line sizes bigger than a few bytes are more efficient in terms of memory resource usage being able to hold more complex data structures than only basic data types. But they also are more complex in construction resulting in increased access latencies and area requirements because of their need to repeatedly initiate memory transfers until the line be completely filled. On the other hand, cache memory resource usage can also be improved by increasing the so-called associativity, which in term raises the overall cache complexity.

Replacement and allocation strategies define how the cache memory resources are (re-)used whereas the write-back strategy targets memory access "beyond" the current cache level. The replacement strategy contributes to the overall efficiency of memory use to a bigger extent than the other caching strategies. Frequently, write-allocate is used in conjunction with write-back, while write-through performs better with no-write-allocate[12].

C.1 Reconfiguration Effects

It is vital that run-time reconfiguration leave the cache infrastructure in a sane state. This, of course, can be achieved by a forced invalidation (preceded by a write-back of dirty lines), but for obvious reasons unnecessary invalidation has to be avoided. We will therefore first elaborate on the types of reconfiguration and their implications for the already cached data.

If not explained otherwise, in the following discussion increasing or decreasing means doubling or halving the respective number.

C.2 Changing Associativity with Constant Number of Lines

We first show that it is possible to increase associativity at the cost of a decreased number of sets while keeping a correct cache state. This can be achieved by means of reordering, as constructively proven below.

Let $n = 2^m$ be the cache associativity for $m \in \mathbb{N}_0$, let s be the number of sets, a the address size in bits, b the line size in bytes, and t the tag vector in bits.

For a one-way associative, i.e. direct-mapped, cache, we therefore have $a = t + \mathrm{ld}(s) + \mathrm{ld}(b)$. Using $p = \mathrm{ld}(s)$ and $q = \mathrm{ld}(b)$, this can be written as $a = t + p + q$ or $a - q = t + p$. Setting $d = a - q$ leads to $d = t + p$ —the tag length and the set address.

Going from an n-way ($m > 0$) associative cache to a $2n$-way associative, the necessary doubling of the associativity at constant line size results in the set address size being decreased by exactly one bit, as the number of sets was implicitly halved. Correspondingly, a tag size increase of one bit occurs, hence $d = t + 1 + p - 1$.

For a given line with address d' let $d' \equiv k \pmod{s}$ and w.l.o.g. $0 < k < \frac{s}{2}$ and $k \equiv 0 \pmod{2}$, and another line $d'' \equiv l \pmod{s}$ with $l \neq k$ and $l \neq k + \frac{s}{2}$, we therefore get $d' \equiv k \pmod{\frac{s}{2}}$ and $d'' \equiv l \pmod{\frac{s}{2}}$ or $d'' \equiv l - \frac{s}{2} \pmod{\frac{s}{2}}$. Consequently, different lines are stored in different sets.

For checking the second half of cache lines starting with line $\frac{s}{2}$, the restriction of $l \neq k + \frac{s}{2}$ does no longer apply. Therefore, for a third cache line $d''' \equiv k \pmod{\frac{s}{2}}$, $t_{d'} = t_{d'''}$ is possible and valid.

Accordingly, $d''' \equiv k + \frac{s}{2} \pmod{s}$ applies and we get the first bit of the set address $p_{d'}(0) = 0 \neq p_{d'''}(0) = 1$, resulting in a new tag address $t^{new}_{d'} = t_{d'} \;\&\; p_{d'}(0) \neq t^{new}_{d'''} = t_{d'''} \;\&\; p_{d'''}(0)$, i.e. the new line can be stored in the very same set ("&" indicates concatenation of the bits).

Looking at one set, in the general case of $n = 2^m$, with $m \in \mathbb{N}, \forall\, i \in \{1, \ldots, n\}$, we have all addresses $d^i \equiv k \pmod{s}$, thus $t_{d^i} \neq t_{d^j}\ \forall\, i \neq j$. Likewise, addresses $\equiv l \pmod{s}$, $l \neq k \ \wedge l \neq k + \frac{s}{2}$ will not be mapped to the same set.

For addresses $e^{(1)}, \cdots, e^{(n)}$ of a different set having $e^i \equiv k + \frac{s}{2} \pmod{s}$ and $e^i \equiv k \pmod{\frac{s}{2}}$, we therefore get $t_{e^i} \neq t_{e^j}$ for $i \neq j$. So, for the first bit of the set address, $p_{d^i}(0) = 0 \neq p_{e^j}(0) = 1$ applies for any given i, j. Adding this bit to the former tag address, the new tag addresses therefore become $t^{new}_{d^i} = t_{d^i} \;\&\; p_{d^i}(0) \neq t^{new}_{e^j} = t_{e^j} \;\&\; p_{e^j}(0)$ meaning that they all can be stored safely in the new, double-sized set.

It must be noted that this reconfiguration cannot be performed sequentially without requiring an additional half-sized memory. So, the designer has the choice on whether to first back the rear half into the additional memory, then rearrange the front half and finally add the backed rear half, or to first flush and write-back the rear half and then only rearrange the front half. The second possibility clearly reveals that a trade-off between minimum chip area (used for reconfiguration logic) and data preservation has to be made.

Decreasing the associativity with a fixed number of lines is only possible if each set contains the same amount of lines with odd and even tag. Even then, the problem remains that successive reordering is not possible. Therefore, the same scheme of flushing (or backing) the rear cache half and—starting with the first line proceeding to the end—transferring

just one half of a set's lines from the front half into the rear half is proposed as trade-off.

C.3 Changing Number of Lines with Constant Number of Sets

Changing the number of lines while keeping the number of sets constant directly affects associativity: increasing the number of lines results in double set size and therefore double associativity. The cache needs to be reordered by successively moving the blocks to their new position.

Decreasing the number of lines inevitably leads to forced displacement of half a set's lines. It is sensible to apply an LRU strategy here so that the most recently used cache lines remain stored within the cache.

C.4 Changing Number of Lines with Constant Associativity

Keeping associativity while changing the number of lines results in a change of the number of sets and is equivalent to adding empty cache lines. If addition takes place in powers of two, the redistribution of cache lines is easy, i.e. odd lines (their tags end with '1', they belong to the new rear half) are moved from the original "front" lines to the newly added "rear" lines and the old line is invalidated. Because associativity remains the same and due to the fact that a potentially added set can store the same number of lines as the "old" sets, the reconfiguration is also possible with sets containing entirely odd or even tags. Of course, lines with even tags are not moved as they already are where they belong to. In any case, the tag has to be shortened due to the larger amount of sets.

When decreasing the number of lines, again in powers of two, each removed line from the rear part of the cache might have a corresponding "antagonist" in the front half, one of them both having to be displaced. Ideally, an unmodified cache line would be flushed in order to save effort. In addition, the tag is prolonged by one bit to distinguish between former front or rear position just as explained above. Again, a good trade-off is to flush the rear half and only modify the remaining front part.

C.5 Changing the Replacement Strategy

Change of replacement strategy has no direct effect on the stored data, but only affects the quality of data caching. However, it must be taken into account that switching the replacement strategy might introduce some information loss regarding the reuse statistics of cache lines.

Replacement strategies form an implicit hierarchy: on lowest level, we find random replacement, which does not keep track of previous line accesses. With pseudo-random replacement, a global counter register is introduced. More information is given by the FIFO strategy, which knows the storing order of each line per set. On top, we find pseudo-LRU and LRU strategies, providing knowledge of cache-line-individual usage history.

It is trivial to change from a more sophisticated to a simpler strategy; for this way, only minor housekeeping has to be done such as setting the counter register to a random value for pseudo-LRU, or entering the least recently used line into the FIFO register.

Going the other way suffers from a lack of information quality. The simpler strategy cannot provide replacement information as detailed as would have been possible with the newly chosen strategy.

For the first series of measurements, however, we did not reuse any previous information at all leading to only slightly worse results concerning cache line replacement and significantly decreasing the size of the reconfiguration controller.

C.6 Changing the Write Strategy

When switching off write-allocation, subsequent read or write accesses to the cache need to take care of eventually previously modified contents. In this case, the old information still has to be written back into memory.

However, information is not lost, because for any stored and modified data, the "Modified" bit is set accordingly and therefore upon later displacement of the modified cache line, the information will be written back.

When changing from write-back to write-through, it might also occur that dirty lines have not been written back to memory yet.

Again, checking the "Modified" bit is done in any case before accessing a cache line in order to care for regular write-after-read accesses, so no additional logic has to be implemented for guaranteeing data consistency after any parameter change concerning write strategy. Hence, offering the possibility to change write-access strategies does not impose any side effects.

C.7 Resource Usage and Time Consumption

The reconfiguration controller currently takes 420 Slice Flip Flops and 12,068 LUTs. It can run at a maximum frequency of 10.05 MHz. These numbers reflect the current state of the implementation, which has not yet been technology mapped. Though, we are confident that an optimized version will lead to significant speed-up.

Reconfiguration of parameters such as write-strategies only requires to set a value in a dedicated register, which consumes four cycles. When changing the replacement strategy, for each set the replacement information is updated. This requires a basic setup time of two cycles, then one cycle per set to clear the information and one final cycle to signal the successful end. Thus, for a cache with 1024 sets, 1027 cycles are needed.

Increasing the associativity also needs a basic setup time of two cycles. For each line of the "rear" half, the "Modified" bit is checked and, when modified, the line is written back. Additionally, the line is totally cleared. All in all, the processing time per line is two cycles in the best case, and twelve cycles in the worst case. After having cleared the "rear" half, one synchronization and setup cycle is inserted. Each line of the "front" part is now moved to its new location within three cycles, and two final cycles end the reconfiguration process. Thus, for 1024 lines, 2+512*2+1+512*3+2 = 2565 cycles are needed in the best case, while the worst case requires 2+512*12+1+1512*3+2 = 7685 cycles.

Few additional cycles are added by protocol overhead. See Section D.5 for details.

D RCA: Reconfigurable Cache Architecture

The proposed reconfigurable cache architecture (RCA) enables run-time changes of several cache parameters as outlined in this section. However, due to hardware implications, the maximum value of certain design parameters—such as e.g. overall cache size or the maximum amount of associativity—is already set before synthesis.

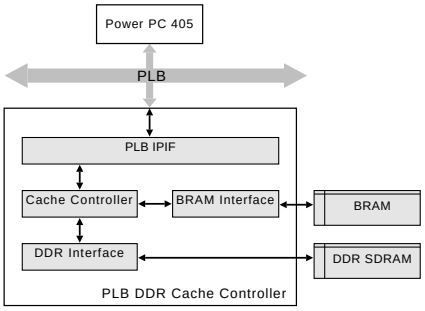

Figure 1. Basic Cache Controller Setup

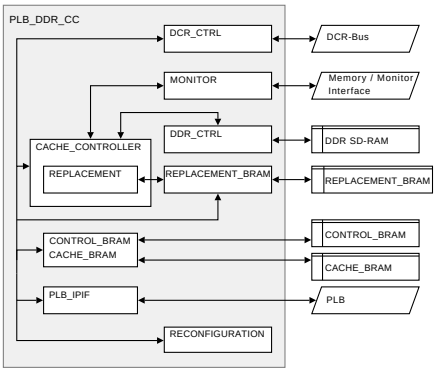

Figure 2. Reconfigurable Controller: Architecture

D.1 Cache Controller: Basic Architecture

For our Reconfigurable Cache Architecture, we developed a cache controller from scratch, merging it with the proven, working designs provided by the Xilinx FPGA development software. Our basic setup consists of a Power PC processor embedded into the FPGA, the CoreConnect[13] bus interface technology (Processor Local Bus, PLB), and a DDR SDRAM controller (PLD DDR Controller V1.11a)[14] as shown in Fig. 1. CPU memory accesses appear on the PLB and therefore are processed by the cache controller, which will then correspondingly access the cache and external SDRAM used as main memory.

Our targeted hardware platform[15] supplies only two 64M*16 DDR SDRAM controllers, i.e. with every clock cycle 2*32 bits can be written to or read from memory. Thus, we use the standard PLB data width of 64 bits resulting in a preferred cache line width of 64*n bits, where n is usually set to 1 in order to avoid additional latencies caused by memory transfers.

The FPGA's BlockRAM (BRAM) provides fast cache memory and is additionally used to store housekeeping data.

Figure 2 shows the internal architecture of the cache controller. We decided to split the BlockRAM resources into individual, physically separated chunks—holding cache data, control and replacement information—instead of using one monolithic memory area. These chunks are kept separate with respect to reconfiguration, higher throughput, and clock rates although that leads to multiple instances of BlockRAM interfaces.

Reconfiguration is achieved by means of device control registers (DCR), which are accessed via the DCR bus, part of the CoreConnect specification. A rather complex state machine controls the overall functionality of the cache controller. In addition, monitoring functionality is embedded with respect to run-time analysis of memory and cache accesses.

In the following sections, we provide a more detailed description of the individual building blocks of the Reconfigurable Cache Architecture.

D.2 State Machine

The state machine consists of four basic parts for handling read requests, write requests, replacement information, and, finally, reconfiguration. The request handling parts have sev-

eral states for initiating the request processing, handling write-back, and data transfer. The reconfiguration part, which manages cache reconfiguration and ensures cache coherency, is only entered when no cache requests are pending. More specifically it is enforced that all pending write accesses from cache to memory be finished prior to reconfiguration.

D.3 Replacement Strategy

In our design, we provide various replacement strategies ranging from the trivial random and FIFO strategies to the rather complex LRU strategies.

In the FIFO case, only a single register is required per set in order to keep track of the last written line number. Prior to writing a line, this register is increased by one modulo the cache associativity.

LRU and Pseudo LRU, in term, require use of additional BlockRAM area to store the per-set access bits, for instance 28 bits for one line of an 8-way set-associative cache when using LRU, or 10 bits per set when using pLRU. Due to this enormous chip area usage, we narrowed the configuration space to a maximum associativity of eight when using LRU or pseudo-LRU.

D.4 Monitoring

Both of two available modes track memory access characteristics by writing the collected information in a consecutive stream to the monitor output register, which in turn can be stored in a dedicated BlockRAM area by a custom monitoring information handler. Once that area would be filled, an interrupt might be risen so that the monitored data could be transferred to external memory and further processed.

The monitor output register decodes the affected cache line number within a set, information regarding transfer mode, whether the transfer was a hit or a miss, four bits indicating the byte in the cache line access, and whether the collected information can be regarded as valid. In the second mode, 30 bits of the corresponding memory address are additionally tracked.

D.5 Cache Reconfiguration

Reconfiguration is possible through a simple register interface accessible via the DCR bus. This way, the introduction of user-defined instructions and use of the APU controller is avoided. Figure 3 depicts how the processor, the controller and the reconfiguration mechanism communicate.

The interface basically consists of a single control/status register, and seven dedicated configuration registers responsible for the individual configurable values.

Configuration takes place by writing the appropriate value into the corresponding configuration register. The end of a reconfiguration process is signalled by sending a “Done” signal to the control/status register.

D.6 Simulation Results

Our architecture was evaluated running a test program. The hardware parameters for the tests are 4096 cache sets, a data width of 64 bits, and, accordingly, a line size of 8 bytes.

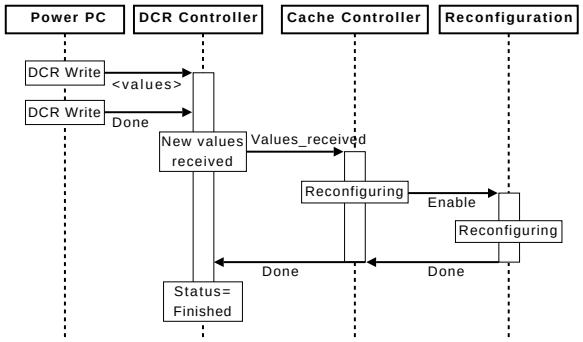

Figure 3. Components involved in reconfiguration process and their communication behaviour.

Associativity	Write-Allocate	Write-Back	Replacement	Run Time
1	$\surd$	WT	–	36,525 ns
1	$\times$	WT	–	36,600 ns
1	$\surd$	WB	–	36,750 ns
1	$\times$	WB	–	36,850 ns
2	$\surd$	WB	pLRU	37,000 ns
4	$\surd$	WB	pLRU	36,525 ns
4	$\surd$	WB	FIFO	36,575 ns
4	$\surd$	WB	LRU	36,825 ns
4	$\surd$	WB	pRandom	38,425 ns
8	$\surd$	WB	pLRU	38,325 ns
16	$\surd$	WB	FIFO	40,475 ns
16	$\surd$	WB	pRandom	40,250 ns

WB=Write-Back, WT=Write-Through

Table 1. Simulation Run Results

This pseudo-code was manually translated into PPC assembly language in order to ensure that no compiler code optimization or rearrangement take place. With this test program, we simulate initial writes of multiple cache lines, overwriting of cache lines with and without replacement, cache hits on stored and overwritten lines, and cache read and write misses on lines with wrong tag or wrong byte-enable bits.

The results of our simulations are depicted in Table 1, listing the run-time for our test program for various cache configurations. It must be noted that the aim of this test program was to mainly validate proper functionality and perform debugging, where required.

With our simple test routine, we were able to validate our hardware design and to successfully demonstrate changing cache hardware parameters such as cache associativity, write allocation, as well as write-back and replacement strategy. Note that the raising times with higher associativities are due to non-parallel implementation of the different ways, which makes the reconfiguration process a lot easier.

E Toolchain Integration

Configuration of the architecture takes place within boundaries defined before the synthesis process. These boundaries are specified by a set of generics, which determine the most complex case for later run-time reconfiguration; for instance, setting the maximum replacement strategy to "pseudo-random" for synthesis will effectively disable later use of "superior" strategies like LRU because the required resources won't be synthesized.

In order to ease configuration editing, we developed a configuration tool, which tightly integrates into the Xilinx XPS work flow. The user enters the desired maximum configuration and desired timing behaviour, based on which the project data files will be altered accordingly.

F Conclusion and Outlook

The described cache architecture has been implemented and thoroughly simulated. We thereby proved theoretically and practically that a reconfigurable cache architecture is feasible and that it is indeed possible to achieve run-time reconfiguration while keeping the overall cache in a sane state without a mandatory need to entirely flush and reinitialize the cache memory and controller.

The design is currently synthesized into hardware to match the Xilinx ML310 and ML403 evaluation boards in order to get numbers regarding additional hardware and timing costs compared to a standard DDR SDRAM controller.

In a next step, the hardware infrastructure will be tightly integrated into our existing monitoring and visualization tool chain for data locality and cache use optimization. Furthermore, we are investigating on further possible run-time reconfiguration changes like cache line size, dynamically providing more BRAMs and exploiting them by unifying them with the already existent cache memory. Besides, we plan on creating a run-time library for use by both operating systems and applications themselves allowing them to individually fine-tune their caching system.

Once fully implemented, the architecture is supposed to speed up long-running applications by means of cache reconfiguration with the cache parameters being determined at run-time by a monitoring environment both in hardware and at operating system level. Of course, excluding the monitoring components and having the program reconfigure the cache itself, is also possible.

In a side-project, cache partitioning is explored. By partitioning the cache, serving memory-hungry applications better than short-term applications should be possible for multitasking systems.

References

1. J. Tao and W. Karl, *Optimization-oriented visualization of cache access behavior*, in: Proc. 2005 International Conference on Computational Behavior, LNCS vol. **3515**, pp. 174–181, (Springer, 2005).
2. A. Gordon-Ross, C. Zhang, F. Vahid and N. Dutt, *Tuning caches to applications for low-energy embedded systems*, in: Ultra Low-Power Electronics and Design, Enrico Macii, (Ed.), (Kluwer Academic Publishing, 2004).

3. A. Gordon-Ross and F. Vahid, *Dynamic optimization of highly configurable caches for reduced energy consumption*, Riverside ECE Faculty Candidate Colloquium, Invited Talk, (2007).
4. R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu and S. Dwarkadas, *Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures*, in: MICRO 33: Proc. 33rd annual ACM/IEEE international symposium on Microarchitecture, pp. 245–257, (ACM Press, New York, 2000).
5. A. Gordon-Ross, F. Vahid and N. Dutt, *Automatic tuning of two-level caches to embedded applications*, in: DATE '04: Proc. Conference on Design, Automation and Test in Europe, p. 10208, (IEEE Comp. Soc., Washington, DC, 2004).
6. C. Zhang and F. Vahid, *Cache configuration exploration on prototyping platforms*, in: IEEE International Workshop on Rapid System Prototyping, p. 164, (IEEE Computer Society, 2003).
7. A. Gordon-Ross, F. Vahid and N. Dutt, *Fast configurable-cache tuning with a unified second-level cache*, in: ISLPED '05: Proc. 2005 International Symposium on Low Power Electronics and Design, pp. 323–326, (ACM Press, New York, 2005). `http://portal.acm.org/citation.cfm?id=1077681`
8. A. Gonzàlez, C. Aliagas and M. Valero, *A data cache with multiple caching strategies tuned to different types of locality*, in: ICS '95: Proc. 9th International Conference on Supercomputing, pp. 338–347, (ACM Press, New York, 1995).
9. C. Zhang, F. Vahid and W. Najjar, *A highly configurable cache architecture for embedded systems*, in: ISCA '03: Proc. 30th Annual International Symposium on Computer Architecture, pp. 136–146, (ACM Press, New York, 2003).
10. T. Sherwood, E. Perelman, G. Hamerly, S. Sair and B. Calder, *Discovering and exploiting program phases*, IEEE Micro, **23**, 84–93, (2003).
11. A. J. Smith, *Cache memories*, ACM Computing Surveys (CSUR), **14**, pp. 473–530, (1982).
12. S. M. Müller and D. Kröning, *The impact of write-back on the cache performance*, in: Proc. IASTED International Conference on Applied Informatics, Innsbruck (AI 2000), pp. 213–217, (ACTA Press, 2000).
13. IBM, *Coreconnect architecture*, (2007). `http://www-03.ibm.com /chips/products/coreconnect`
14. Xilinx, Inc., *Plb double data rate (ddr) synchronous dram (sdram) controller – product specification*, (2005). `http://japan.xilinx.com/bvdocs/ipcenter /data_sheet/plb_ddr.pdf`
15. Xilinx, Inc., *Xilinx development boards: Virtex-4 ML403 embedded platform*, (2007). `http://www.xilinx.com/xlnx/xebiz/designResources /ip_product_details.jsp?key=HW-V4-ML403-USA`.